Gesture

A Cyberphysical system to interpret American Sign Language

Scott Tyler Strobel Sievert

Submitted under the supervision of Jarvis D. Haupt to the University Honors Program at the University of Minnesota-Twin Cities in partial fulfillment of the requirements for the degree of Bachelor of Science in Electrical Engineering, *cum laude.*

June 2015

Abstract

Language obstacles can often exist when hearing impaired individuals interact with others who are not well-versed in sign language. We propose and develop a system that utilizes commercially available, wireless armbands that are capable of detecting acceleration, orientation and electromyography (muscle activity) data. Our interpretation system is designed to translate gestures in a "word-by-word" manner by assuming that each sign language sign corresponds to a unique English word, which eliminates inter-word dependence and makes this a classification problem. Using convolutional neural networks as our interpretation algorithm, we achieve high classification accuracy of roughly 90% for a dictionary of 7 words as well as a "null" word.

Contents

List of Figures

1 Introduction

A significant proportion of the American population is hearing impaired, approximately 8% [2]. They face a language obstacle and may require interpretation from American Sign Language to English or vice versa. They may require human interpreters, hearing aids or other means of conveying a message.

Our goal is to develop a system that automatically interprets from American Sign Language (ASL) to English, which is called automatic sign language recognition (ASLR). We envision the user signing naturally while our system interprets the signed ASL to English. Ideally, the user would sign naturally as they normally would, with no modifications for interpretation. This means that they would choose signs as they normally would and the interpretation algorithm should adapt to sign length, natural pauses and speed changes that are natural in ASL.

We want our system to be accessible from anywhere (i.e., while walking, sitting or standing, etc). We envision the system being implemented on a mobile device, similar to the system shown in Figure 1. The system described above is inspired by a fictional video titled "Google Gesture" which had been released in the summer of 2014 on Vimeo.[1]

Besides pausing and sign length, several other larger challenges exist while interpreting ASL. This includes the fact that ASL and English sentence structure is different, ASL signs may depend on context and regional differences are present in signing [3]. Besides interpretation details handling context and regional differences, this means that the interpretation system can not be real time because the sentence structure of ASL and English is different. There must be some delay to provide the interpretation algorithm with enough information to interpret a sentence accurately.

Algorithmically, the largest challenge is that ASL signs may depend on context and previous signs. This means that one sign may have many different meanings and a meaning is selected by previous signs, context and facial gestures as well as other factors. For example, some ASL signs have over 100 different meanings and the meaning is selected by previous words or context aka surroundings, who is speaking, etc. This inter-word dependence presents a challenge when devising interpretation algorithms.

[1] https://vimeo.com/97528184

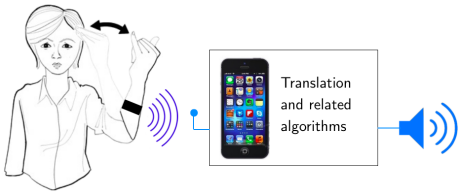

Figure 1: The proposed system utilizes wearable armbands that communicate wirelessly with a mobile device which will implement automatic sign language recognition.

Figure 2: Example of the input system that receives sign language data used by Kadous *et al.* [1] to provide automatic sign language recognition. These gloves provide orientation and limited finger movement data.

2 Previous work

Interpreting sign language is an attractive goal and many systems have been built explicitly with this goal in mind. This has happened with many languages including including American, Chinese, Japanese and Greek and their corresponding sign languages.

These systems are summarized in [3]. This summary includes the algorithmic approach, input methods, type/numbers of participants as well as the accuracy rate. All of these systems are varied in their approaches and no system described takes the same approach as another described system.

The algorithmic approach of these systems seem to follow a similar path. It seems that the teams implementing these systems first develop algorithms suitable to speech recognition (e.g., using hidden Markov models [4]). They then adapt these algorithms to include some complications of sign language as described in Section 1. The choice of adaptations are varied.

The inputs to this system typically fall in two categories, either wired gloves (typically one handed) or video. Examples of such input systems are shown in Figure 2. Such systems tend to have certain restraints. Glove based systems tend to require wearing gloves with delicate components and other systems require the use of an external video camera.

The training among these systems again also varies. A majority of systems trained

and tested on the same individual, while some gathered a few training participants (up to 5) and a few testing participants (up to 5). A notable exception is [5] which gathered 20 training and testing participants. Gathering many participants introduces variation in signs performed. Regional, cultural and personal differences in sign language are present and gathering more participants to provide input data helps protect against these differences.

The results from these systems tend to do well in certain scenarios. For example, one system could recognize seven different words from three different categories with 92-96% accuracy by using hidden Markov models [6]. Other systems had much smaller vocabularies and/or less inter-word dependence and/or many other factors including differences in accurately predicting even in the presence of regional sign differences.

Most of these systems rely on hidden Markov models (with various modifications/adaptations). Hidden Markov models are heavily used in speech recognition and take in natural speech data and classify each spoken word. It is assumed that there is little inter-word dependence or that there is a clear distinction between words.

3 Proposed System

3.1 Input

One of the more challenging problems in ASLR is obtaining the input data. As illustrated in Section 2, other ASLR system have built custom solutions including wired gloves or use video. These gloves give data about the finger position as well as (possibly) acceleration and orientation while video gives information about hand movement and hand velocity. These systems typically have several restraints such as being seated in front of a camera or wearing clothing with sleeves.

We have decided to use the Myo armbands,[2] developed by Thalamic Labs, Inc. These armbands are worn on the upper forearm and report acceleration, orientation and finger movement data (through surface electromyography or sEMG). They can communicate via Bluetooth 4.0 to smartphones and computers. We plan for the user to wear two of these armbands to receive data about both arms.

This means that at the broadest level, the ASLR system we describe is possible. We can receive information about the gestures the user is performing. The overview of this system is described in Section 3.2 and the details of the system are described in Section 3.3.

[2] https://www.thalmic.com/en/myo/

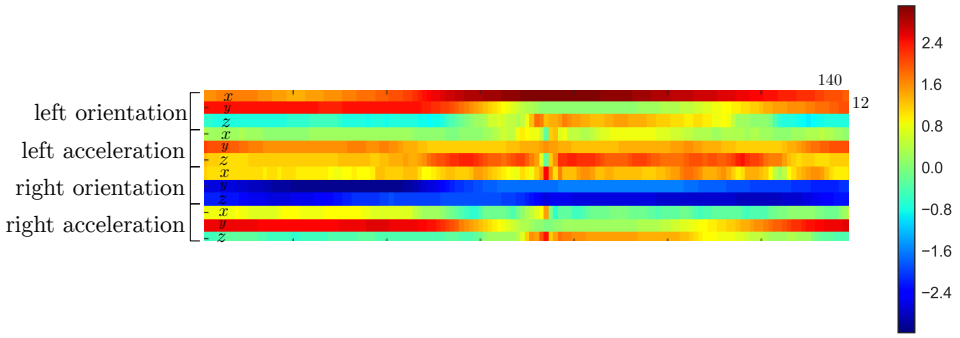

Figure 3: A visualization of 140 samples of the data jointly produced by the two Myo armbands. Each provides acceleration and orientation data in three axes, resulting in 12 channels of data.

3.2 Overview

Our overall system has Myo armband data as an input and computer data as an output. The early release of the armbands[3] dictated that we use iOS and the armbands could only supply acceleration and orientation information. After recording this data and transferring it to a computer, it could be interpreted by the interpretation algorithms described in Section 5.

The Myo armbands currently support acceleration, orientation and sEMG data. The acceleration and orientation is collected in each of the three spatial dimensions and the sEMG data is collected via 8 sensors on the armband. However, as these are new products, and the early releases we were using did not enable sEMG data. This meant that we only received acceleration and orientation data in three dimensions on two arms.

We can think of these 12 inputs ({acceleration, orientation}, {left, right}, $\{x, y, z\}$) as a 12-dimensional vector. We can then stack these vectors into a $12 \times n$ matrix. For this use case, we choose $n = 140$. Because the armbands report acceleration and orientation events at a rate of 50 Hz, this means we capture 2.8 second window. This is ample time to complete a single sign and the interpretation algorithm should be designed to accept multiple signs in this window. An example is shown in Figure 3

3.3 Details

In this section we describe the details of our system and some of the more specific elements of our system. While it is described here, our entire implementation can be found online

[3]We obtained a developer-only beta in October 2014

on Github[4]. This includes all the software and data necessary to run our implementation (lacking only in hardware).

3.3.1 Communicating with Myo armbands to acquire input data

The Myo armbands communicate to a smartphone via Bluetooth 4.0. They send alerts to the receiving Bluetooth device or phone in the form of notifications which run a certain function upon receiving an notification for a acceleration or orientation event.

In this report, we will include the core of the system used to communicate with the Myo armbands in Appendix D although the entire project/app can be found on Github. We had to use global variables because of the different function calls for acceleration and orientation and also had to hardcode the name of armband so we could communicate with *two* armbands.

It should be noted that this code assumes the Myo armbands are running an specific version of the beta firmware. This code compiled using XCode 6 (*not* a beta release!). The specific betas (as raw binary files) the armbands are assumed to be running can be found on Github. If they're not running this firmware, the iOS app fails to communicate with the Myo armbands.

When the phone receives notification that it as received an acceleration or orientation event, it obtains the acceleration/orientation in the x, y and z directions. It then puts those three numbers in the appropriate global variable and increments a counter. When this counter reaches 140, it resets the counter and writes this matrix to a file in a CSV format. This CSV file represents a 3×140 matrix for one particular data type (e.g., right orientation). The three rows of this matrix represent the three spatial dimensions.

3.3.2 Data processing approach

The iOS interface writes files such as `teacher_left_acceleration.csv`. These files are transferred to a computer via the iTunes Connect interface and interpreted by a computer. The computer program (written in Python and shown in Appendix C) reads in 4 different files corresponding to left/right and orientation/acceleration then stacks these 3×140 matrices into a 12×140 matrix.

After a matrix is made, we have to format the matrices for the interpretation algorithm. We stack the matrices into a three-dimensional array and assign corresponding a label

[4]Currently a private repository but can (eventually) be found at https://github.com/scottsievert/gesture

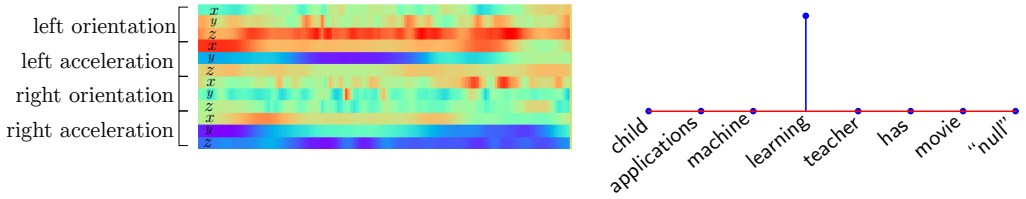

Figure 4: An input pair to the interpretation system. The input matrix represents the 12 channels of received by the Myo armbands and the output vector has a single nonzero entry taking the value 1 at location corresponding to the word spoken. If we have n words, the associated output vector is of length $n + 1$ as we have a "null" word.

corresponding to `word` to each input matrix. An example of one such pair is shown in Figure 4. Complete details including the raw input data can be found on Github but the Python script to format this data can be found in Appendix C.

After we have stacked this data into a three dimensional array, we use `scipy.io.savemat` to save a MATLAB `.mat` file. We load this file into MATLAB for the interpretation algorithm.

4 Training

Before we start on the interpretation algorithm, we have to provide data. The machine needs data to train so it knows which input matrices correspond to a particular word and it also needs data to test on each word. These two classes of data will be referred to as "training data" and "testing data" respectively. Collectively, we'll refer to "data" when referring to the collected data as a whole.

The method we have selected to collect data is rather simple and works through an iOS device as described in Section 3. After having the user pair the armbands with the iOS device and some short setup, the start the data collection process. This process aims to provide as natural data as possible and not require the user to move their arms for other purposes than to make a sign (i.e., reaching for their phone to say "done with word").

In our training process, we have the phone beep every 3 seconds. Upon hearing this beep, the user knows to make a set of gestures corresponding to a particular word and does

so. After they have done so, they can perform any action (e.g., writing on a chalkboard).

In this process, we train 8 different gestures. These gestures correspond to 7 different words as well as a "null" word where nothing is being performed. We collect 100 different measurements of each sign for a total of 800 different training data (input/output) pairs.

4.1 Testing and training data

We have to separate the training and testing data from the collected instances of each word. Our data is 800 word pairs (denoted by m) and

$$m = [w_{1,1:100}, \quad w_{2,1:100}, \quad \cdots, \quad w_{8,1:100}]$$

where $w_{n,1:100}$ corresponds to 100 different word pairs for word n. We decide to randomly permutate m and then use 40%/60% for training/testing data or

$$\begin{aligned} \text{training data} &= \text{random\_permutation}(m)_{1:320} \\ \text{testing data} &= \text{random\_permutation}(m)_{321:800} \end{aligned}$$

For example, $\text{random\_permutation}(m)_{1:4}$ could be

$$\text{random\_permutation}(m)_{1:4} = [w_{8,6}, w_{7,5}, w_{3,9}, w_{4,29}]$$

These random permutations use identical seeds so words and the corresponding labels are always matched. As shown in Section 6 we see this random permutation balances the inputs fairly well. Different words have approximately equal numbers in testing and training data.

5 Interpretation

During interpretation, we decided to make several simplifying assumptions to make this interpretation problem more tractable. We assume that no inter-word dependence exists and each sign corresponds to a unique word. Additionally, we assume that the sentence structure of ASL and English are the same.

By eliminating the inter-word dependence found in sounds language, we make this a classification problem. This simplified problem is similar to speech recognition; given some input data, what word (either with ASL or speech data) is the user trying to make? There have been systems developed that rely on the same assumptions and process sign language data using multi-modal hidden Markov models [4].

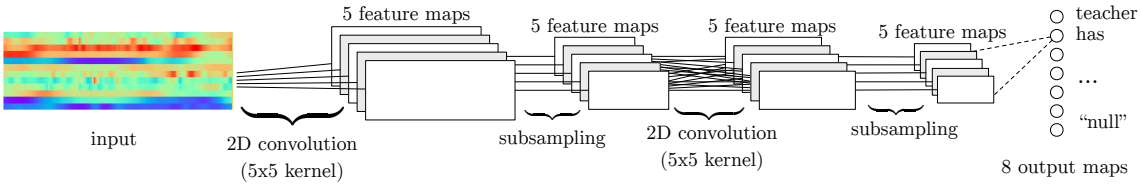

Figure 5: A depiction of the convolutional neural network utilized to perform the word-level classification.

Our choice for interpretation algorithm is convolutional neural networks (CNNs) using a pure Matlab toolbox, the DeepLearnToolbox[5]. These networks are invariant under translation, scale, rotation squeezing and stroke width [7]. These CNNs are meant to recognize unique labels from an input matrix and are translation invariant. This means that when the signal is shifted in time, the output remains constant (assuming that the entire sign is in the event window).

CNNs involve a neural network. Neural networks consist of a number of nodes or neurons as they model biological computation. Each one of these neurons produces an output by taking a linear combination of the inputs and using that as input to some nonlinear function or

$$y_k = f\left(\sum_i c_i x_i + b\right)$$

where y_k is the output of one node, n inputs x_i, some constants c_i and a bias term b as well as some nonlinear function f (typically a sigmoid or "s-shaped" function). Neural networks may have many layers. That is, the input may be fed into a set of nodes (as described by y_k above). If the network has more than one layer, this is fed into another layer set of input nodes (against similar y_k).

The constants c_i and b are found via training. During training, we both know the target result and the input data. This means that while training, the neural network uses feedback or back-propagation to tune c_i and b. Because it knows the correct result, it can optimize c_i so when the training input is fed in it produces the correct result. The training defines how well a neural network works and is where most of the development energy is focused.

As we were using a software package that implemented CNNs and their training for us, the training only had several parameters to tune. This included the number of layers, the number of nodes at each layer, the step size and two parameters that indicated how long the training would take. We explored many different configurations with a different number

[5] https://github.com/rasmusbergpalm/DeepLearnToolbox

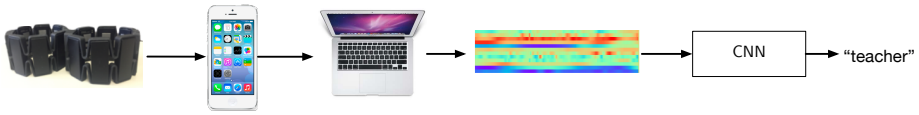

Figure 6: A visualization of the data flow in our system. We collect data from the Myo armbands via an iOS device, process that data on a computer to produce an input matrix and feed that to an interpretation algorithm.

of layers and a different number of feature maps per layer.

As the training process uses gradient descent, it is important to consider the step size. If the step size if too large, the neural network training algorithm may choose a set of constants that places the result in a region of the sigmoid function where the magnitude of the derivative is small. Because the neural network operates off gradient descent, the training algorithm relies on the magnitude of the derivative greater than being large enough in magnitude to provide effective optimization steps. After selecting an appropriate step size, we found that we could obtain high accuracy with a CNN that had two hidden layers and 5 feature maps in each hidden layer.

6 Results

We have provided an entire input system to allow interpretation from Myo armbands using any interpretation algorithm. This system provides communication between the Myo armbands and an iOS device that collects acceleration and orientation information, moves the raw files from the iOS device to a computer, and formats that data into an input matrix. This complete system can be found on Github and a pictorial representation is in Figure 6

As we can see in Figure 7 we have relatively high accuracy even while training with minimal amounts of data. As described in Section 4 we randomly permute the input word pairs and use 40% for training and 60% for training. The testing and training sets are disjoint sets that have no overlap: a particular word pair can not belong to both the training and testing set.

7 Future Work

Immediate work includes expanding the dictionary to include more than 7 words as well as a null word. We'd also like to expand to include input data from multiple users and test natural signing. This will include more personal differences in ASL. Additionally, more

Word	Number of testing signs	Incorrect predictions	Classification accuracy
child	51	6	88.23%
applications	67	8	88.05%
machine	62	6	90.32%
learning	59	7	88.13%
has	63	8	87.30%
movie	62	9	85.48%
teacher	61	5	91.80%
"null"	58	4	93.10%
Total	483	53	89.03%

Figure 7: Table of classification accuracies for a 7-word dictionary (with additional "null" word). Training was performed on 40% of the overall data acquired and testing on the remaining 60%.

distant future work involves allowing some inter-word dependence.

As these Myo armbands are new devices, subsequent functionality has been released. In particular, the raw finger movement data through sEMG is available on iOS.[6] This would allow an additional 8 channels per arm for a total of 28 channels, a significant increase over the 12 channels we currently have.

Other future work includes porting this developed interpretation algorithm to a mobile device. As the developed CNN is of a relatively small size and Matlab-like type frameworks for iOS exist[7], a port to iOS is possible.

8 Conclusion

We have developed a system to allow to allow for automatic sign language interpretation involving the Myo armbands, a iOS app and a Matlab/Python software framework on a computer. Using simplifying assumptions that remove inter-word dependence by assuming each sign corresponds to a unique word this problem of interpretation is tractable. For this classification problem with a limited dictionary, we results with a high classification accuracy. The developed interpretation algorithm can be ported to a mobile device by translating the code of the pure Matlab toolbox DeepLearnToolbox code or using existing mobile device CNN frameworks.

[6] This SDK was released on April 13th, 2015

[7] http://scottsievert.github.io/swix

References

[1] M. W. Kadous *et al.*, "Machine recognition of auslan signs using powergloves: Towards large-lexicon recognition of sign language," in *Proceedings of the Workshop on the Integration of Gesture in Language and Speech*, pp. 165–174, Citeseer, 1996.

[2] J. A. Holt, S. Hotto, and K. Cole, *Demographic aspects of hearing impairment: Questions and answers*. Center for Assessment & Demographic Studies, Gallaudet University, 1994.

[3] S. C. Ong and S. Ranganath, "Automatic sign language analysis: A survey and the future beyond lexical meaning," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 27, no. 6, pp. 873–891, 2005.

[4] T. E. Starner, "Visual recognition of american sign language using hidden markov models.," tech. rep., DTIC Document, 1995.

[5] H. Matsuo, S. Igi, S. Lu, Y. Nagashima, Y. Takata, and T. Teshima, "The recognition algorithm with non-contact for japanese sign language using morphological analysis," in *Gesture and Sign Language in Human-Computer Interaction*, pp. 273–284, Springer, 1998.

[6] A. Braffort, "Argo: An architecture for sign language recognition and interpretation," in *Proceedings of Gesture Workshop on progress in gestural interaction*, pp. 17–30, Springer-Verlag, 1996.

[7] Y. LeCun and Y. Bengio, "Convolutional networks for images, speech, and time series," *The handbook of brain theory and neural networks*, vol. 3361, p. 310, 1995.

A Training algorithm

```
% This script trains the convolutional neural network (covnet or CNN). In this,
% we define several the CNN using the DeepLearnToolbox and separate the training
% and testing data.
%
% PARAMETERS:
%   P_TRAIN    : 0<=P_TRAIN<=1; the porportion of the data we want to use for training
%   LAYER1     : How many feature maps to include in the first  hidden layer?
%   LAYER2     : "                                      " second "           "
%   NUM_EPOCHS : How many training iterations to run. Increasing this will
%                linearly increase the time to train
%   STEP_SIZE  : How large should each optimization step be? Too large and you
%                run into saturation (and low accuracy) and too small you never converge
%                (and also low accuracy)
%
%
% (c) Scott Sievert 2014-2015. All rights resevered.

clc; close all; clear all;
f = make_functions();
P = addpath(genpath('./DeepLearnToolbox'));
P = addpath(genpath('./'));

% should we test the training?
TEST = 1;

% how much of the data do we want to train with?
P_TRAIN = 0.4;
NUM_EPOCHS = 50;
LAYER1 = 5;
LAYER2 = 5;

load formatted_data/x-python
load formatted_data/y-python

%x = x-min(min(min(x)));
%x=x./max(max(max(x)));

[~, N_WORDS] = size(y);
N_TRAIN = floor(P_TRAIN * N_WORDS);

```

```
% must be divisble by BATCH_SIZE
N_TRAIN = 630;
BATCH_SIZE = 5;
STEP_SIZE = 1e-1;

% train
train_y = y(:, 1:N_TRAIN);
train_x = x(:, :, 1:N_TRAIN);
% test
test_y = y(:, N_TRAIN+1:N_WORDS);
test_x = x(:, :, N_TRAIN+1:N_WORDS);

% define our layers (seemingly the hard part!)
cnn.layers = {
    struct('type', 'i') % input layer
    struct('type', 'c', 'outputmaps', LAYER1, 'kernelsize', 5) % convolution layer
    struct('type', 's', 'scale', 1) % sub sampling layer
    struct('type', 'c', 'outputmaps', LAYER2, 'kernelsize', 5) % convolution layer
    struct('type', 's', 'scale', 1) % sub sampling layer
};

% define our step size/batch size/epochs (really, the part that took a long time)
cnn = cnnsetup(cnn, train_x, train_y);
opts.alpha = STEP_SIZE;
opts.batchsize = BATCH_SIZE;
opts.numepochs = NUM_EPOCHS;

% train the machine with the training data
cnn = cnntrain(cnn, train_x, train_y, opts);

% save the machine
% 97.7%, 4 words      : CNN, 30 epochs, 5 layer1, 5layer2, gesture11
% 92.2%, 7 words      : CNN, 30 epochs, 5 layer1, 5layer2, gesture12
% 89.0%, 7 words+null : CNN, 30 epochs, 5 layer1, 5layer2, gesture13
% 90.5%, 7 words+null : CNN, 50 epochs, 5 layer1, 5layer2, gesture13
% 92.3%, 7 words+null : CNN, 50 epochs, 5 layer1, 5layer2, gesture14
% 91.8%, 7 words+null : CNN, 50 epochs, 5 layer1, 5layer2, gesture15 (training w/ 630)
name = sprintf('machines/CNN_%d_epochs_%d_layer1_%d_layer2_%0.2f_stepsize_gesture15',
 ↪  opts.numepochs, LAYER1, LAYER2, STEP_SIZE);
save(name, 'cnn', 'N_TRAIN', 'N_WORDS', '-v7.3');

% test the machine with the testing data
if TEST,
```

```
    [err, bad] = cnntest_orig(cnn, test_x, test_y);
    disp(sprintf('Error on test data: %f', err))

    save('y_bad', 'test_y', 'bad')
end
```

B Interpretation algorithm

```
% This script loads in the CNN that SScnn_train.m wrote and predicts. These two
% scripts don't necessarily have to be separate but it provided a clear
% distinction for training and testing (and then we could load a CNN without
% training it first). Correspondingly, this script is fairly simple. It follows
% the steps below:
%
%   1. Load the raw data (x and y) and training data
%   2. Split up into training and testing data
%   3. Predict off the training/testing data
%   4. Display the mean/std.dev of results to see we're predicting more than one word
%
% (c) Scott Sievert, 2014-2015. All rights reserved.

clc; close all; clear all;
f = make_functions();
P = addpath(genpath('./DeepLearnToolbox'));
P = addpath(genpath('./'));

% load the raw data (all of it)
load formatted_data/x-python % x
load formatted_data/y-python % y
load('machines/CNN_30_epochs_5_layer1_5_layer2_gesture13'); % cnn, N_WORDS, N_TRAIN
disp('Loaded machine')

y_train = y(:, 1:N_TRAIN);
y_test = y(:, N_TRAIN:N_WORDS);
x_train = x(:, :, 1:N_TRAIN);
x_test = x(:, :, N_TRAIN:N_WORDS);

[words, trained] = size(y_train);
[words, testing] = size(y_test);
disp(sprintf('Number of words        : %d', words));
disp(sprintf('Number of words trained: %d', trained));
```

```
disp(sprintf('Number of testing words: %d', testing));

%% ex1 Train a 6c-2s-12c-2s Convolutional neural network
% will run 1 epoch in about 200 second and get around 11% error.
% With 100 epochs you'll get around 1.2% error
rand('state',0)

% training for 100 epochs and testing on test_x(1:1e3) will give you
% percentage_right of 89.3% including shifting

[yhat, err, bad] = cnntest(cnn, x_test, y_test);
disp(sprintf('Percentage right after training (words not sentences): %0.3f', 1-err));

disp(sprintf('Mean of results: %f', mean(yhat)));
disp(sprintf('std. dev of results: %f', sqrt(var(yhat))));
```

C Data processing script

```
"""
This script formats the data for use by the matlab script. It reads in files of
the form `child_0_n_left_accleration.csv` and makes a 12x140 matrix out of
these. It then stacks these matrices into a 3D array and formats the outputs to
be a 2D array of 1's and 0's. It outputs these files as .mat files to be read
by the Matlab DeepLearnToolbox
"""

from __future__ import division
__authors__ = {'Scott Sievert' : 'sieve121@umn.edu', 'Jarvis Haupt' : 'jdhaupt@umn.edu'}

from pylab import *
from pandas import read_csv
from scipy.io import savemat

def read_vector(name, PRINT=False):
    """
    Reads in a "vector". That is, it reads in a 3x140 matrix -- one input channel.

    Assumes `name`+ {left, right} `_` {acceleration, orientation} `.csv` exist
    """
    x = read_csv(name, header=None)
    x = asarray(x).flat[:]
```

```
    x = x[:-1]
    x = reshape(x, (-1, 3))
    if PRINT: print "In read_vector. x.shape =", x.shape
    assert x.shape[0] > 140, "Please be a full vector"
    return x.T

def read_file(name, PRINT=False, sentence=False):
    """
    Makes an entire 12x140 matrix for a single sign. Calls :func:`read_vector` 4 times to
    ↪ stack 4 3x140 input channels.
    """

    data = zeros((12, ZERO_PAD+TERMS+0)) if not sentence else []
    i = 0
    postfix = '' if not sentence else '.csv'
    for hand in ['left', 'right']:
        for data_type in ['acceleration', 'orientation']:
            number = '' if sentence else '_0_'
            filename = name+number+hand+'_'+data_type+postfix
            if PRINT: print filename
            channel = read_vector(filename, PRINT=PRINT)

            if not sentence:
                c = channel[:, ZERO_PAD+1:TERMS]
                upper_limit = min(TERMS, c.shape[1]) + ZERO_PAD
                data[3*i:3*(i+1), ZERO_PAD:upper_limit] = c;
            elif sentence:
                data += [channel]

            i+= 1
    if sentence:
        s = data
        # for some reason I hade to change s[0].shape[1]+4 to s[0].shape[1] + 8.
        # (dunno why)
        data = zeros((12, s[0].shape[1]+8))
        for i in arange(4):
            to_add = s[i][:-2]
            data[3*i:3*(i+1), :to_add.shape[1]] = to_add

    return data

from subprocess import Popen, PIPE
def get_number_of_files(word):
```

```
    """
    How many signs for word "teacher" do we have?
    """
    ls_out = Popen(["ls", "../data/_natural_signs/"], stdout=PIPE)
    ls_out = ls_out.stdout.read()
    ls_out = ls_out.split('\n')
    words = []
    for _file in ls_out:
        if word in _file.split('-'):
            words += [_file]
    n = floor(len(words) / 4.0)
    return int(n)
def make_dataset(words, ALL=False, MIN=1, MAX=8):
    """
    Makes an entire dataset of word pairs. This includes stacking all the 12x140 matrices
    ↪  into a 3D vector and formatting the output words with a "1" at a location
    ↪  corresponding to the specific word.
    """
    x_train = zeros((CHANNELS,ZERO_PAD+TERMS))
    y_train = zeros((words.shape[0], 1))
    for word in words:
        n = get_number_of_files(word)
        if ALL: (MIN, MAX) = (1, n)
        upper_limit = MAX
        for i in arange(MIN, upper_limit, dtype=int):
            postfix = '-%d'%i
            x = read_file(ROOT_DIR+word+postfix)

            target = argwhere(word == words)
            formatted_target = zeros((1,words.shape[0]))
            formatted_target.flat[target] = 1

            x_train = dstack((x_train, x))
            y_train = hstack((y_train, formatted_target.T))

    x_train = x_train[:, :, 1:]
    y_train = y_train[:, 1:]
    return x_train, y_train

ZERO_PAD = 0
TERMS = 140 # at 50Hz. How long is the longest word?
ROOT_DIR = '../data/_natural_signs/' # where are the datasets located
CHANNELS = 12 # {x,y,z} * {left, right} * {orientation, acceleration}
```

```
N_TRAIN = 100 # how many times did we train each word?
ML_TRAIN = 80 # how many points should we train with

words = array(['child', 'applications', 'machine', 'learning', 'has', 'movie', 'teacher',
↪  'null'])

if True:
    print "Processing words..."
    xx, yy = make_dataset(words, MAX=N_TRAIN, ALL=True)

    SEED = 42

    np.random.seed(SEED)
    i = arange(xx.shape[2])
    shuffle(i)
    xx = xx[:, :, i]

    np.random.seed(SEED)
    i = arange(xx.shape[2])
    shuffle(i)
    yy = yy[:, i]

    savemat('formatted_data/x-python', {'x' : xx})
    savemat('formatted_data/y-python',  {'y' : yy})

    # plotting the correlation matrix for the similarites between each word. We
    # would hope to see a block identity matrix
    if False:
        x = array([xx[:, :, i].flat[:] for i in arange(xx.shape[2])])
        phi = corrcoef(x)
        matshow(phi, cmap='terrain_r')
        colorbar()
        gca().set_yticks(linspace(0, 1, num=len(words)) * xx.shape[2])
        gca().set_yticklabels(words)
        gca().set_xticklabels([])
        savefig("word_corr.png")
        show()

if False:
    print "Processing sentences..."
    N_SENT = 15 # number of sentences
    sentences = {}
    for i in arange(N_SENT)+1:
```

```
        sentence = read_file('../data/_single_word_sentences_v3/sentence_%d_' % i,
        ↪  sentence=True)
        sentences['single_word_sent_%d' % i] =  sentence
    savemat('formatted_data/single-word-sentences-python-v3', sentences)
```

D Communication with Myo armbands

```
////  ViewController.m
//
//  Created by Scott Sievert on 10/24/14.
//  Copyright (c) 2014 com.scott. All rights reserved.
//
// This source file belongs to an app that communicates with the Myo armbands
// via Bluetooth 4.0. For the full app, see the [Github repo].
//
// This app is not as clean as it could be; our initial approach meant that the
// user had to pause between words. WORD_START, etc are part of this. For
// the most recent code, see the [Github repo].
//
// [Github repo]:https://github.com/scottsievert/gesture

#import <AudioToolbox/AudioToolbox.h>
#import <AVFoundation/AVFoundation.h>

#import "ViewController.h"
#import <MyoKit/MyoKit.h>
#import "MyoInterface.h"

// which word are we training?
// "null", "machine", "learning", "knowledge"...
#define WORD @"null"

// Our dictionary of words
NSArray * DICTIONARY;

@interface ViewController ()
@property (weak, nonatomic) IBOutlet UITextView *textView;
@property (nonatomic, strong) AVAudioPlayer *audioPlayer;
@end

@implementation ViewController
```

```
35
/// provide notification for the user to make a sign
-(void) playBeep{
    NSURL *url = [NSURL fileURLWithPath:[[NSBundle mainBundle]
                                         pathForResource:@"alert"
40                                       ofType:@"mp3"]];
    self.audioPlayer = [[AVAudioPlayer alloc]
                                  initWithContentsOfURL:url
                                  error:nil];
    [self.audioPlayer play];
45 }

/// run when the loads initially -- only once!
- (void)viewDidLoad {
    [super viewDidLoad];
50  // Do any additional setup after loading the view, typically from a nib.

    // make the words visible on screen
    NSMutableArray* DICT_M = [[NSMutableArray alloc] init];
    [DICT_M addObject:[NSString stringWithFormat:@"junk"]];
55  for (int i=1; i<=N_WORDS; i++){
        [DICT_M addObject:[NSString stringWithFormat:@"%@-%d", WORD, i]];
    }
    DICTIONARY = [NSArray arrayWithArray:DICT_M];

60  NSLog(@"Number of words in dictionary: %ld", [DICTIONARY count]);
    NSString * result = [[DICTIONARY valueForKey:@"description"]
     ↪  componentsJoinedByString:@"\n"];
    self.textView.text = result;

    // connect with the Myo armbands
65  [self connect];

    // Posted when a new orientation event is available from a TLMMyo. Notifications are
     ↪  posted at a rate of 50 Hz.
    [[NSNotificationCenter defaultCenter]
                              addObserver:self
70                               selector:@selector(didReceiveOrientationEvent:)
                                     name: TLMMyoDidReceiveOrientationEventNotification
                                   object:nil];

    // Posted when a new accelerometer event is available from a TLMMyo. Notifications are
     ↪  posted at a rate of 50 Hz.
```

```
75      [[NSNotificationCenter defaultCenter]
                                 addObserver:self
                                    selector:@selector(didReceiveAccelerometerEvent:)
                                        name: TLMMyoDidReceiveAccelerometerEventNotification
                                      object:nil];
80
        // to connect to two armbands
        [[NSNotificationCenter defaultCenter]
                                 addObserver:self
                                    selector:@selector(didConnectDevice:)
85                                      name:TLMHubDidConnectDeviceNotification
                                      object:nil];
        // global variables
        // preallocate our data
        a_right = (float*)malloc(sizeof(float) * N);
90      a_left  = (float*)malloc(sizeof(float) * N);
        o_right = (float*)malloc(sizeof(float) * N);
        o_left  = (float*)malloc(sizeof(float) * N);
    }
    -(void)didConnectDevice:(NSNotification*) notification{
95      // to attach to the second armband
        [[TLMHub sharedHub] attachToAny];
    }
    - (IBAction)translate:(id)sender {
    }
100 -(void)connect{
        /*
         * Connect to the Myo armbands; call the API.
         *
         * Work had to be done to connect to *two* armbands.
105      */
        // present the view controller
        [[TLMHub sharedHub] attachToAny];
        [TLMHub sharedHub].myoConnectionAllowance = 2;
        TLMSettingsViewController *settings = [[TLMSettingsViewController alloc] init];
110     [self.navigationController pushViewController:settings animated:YES];
    }

    - (void)didReceiveMemoryWarning {
        [super didReceiveMemoryWarning];
115     // Dispose of any resources that can be recreated.
    }
    void initializeVectors(){
```

```
    /*
     * Initialize the input channels by ensuring every value is 0.
     */
    // set everything to 0
    // a_left for "acceleration left"
    // o_right for "orientation right"
    ai_left = 0;
    oi_left = 0;
    ai_right = 0;
    oi_right = 0;
    for (int i=0; i<N; i++){
        a_right[i] = 0;
        a_left[i] = 0;

        o_right[i] = 0;
        o_left[i] = 0;
    }
}
- (IBAction)startDict:(id)sender {
    /*
     * We are starting the dictionary of words (to be written to a file)
     */
    NSLog(@"Start dictionary!");
    DICT_START = true;
    DICT_INDEX = 0;
    initializeVectors();
    [self startWord];
}

- (IBAction)startSentence:(id)sender {
    /*
     * We have started a sentence
     */
    NSLog(@"Sentence start!");
    SENTENCE_START = YES;
    initializeVectors();
}
- (IBAction)endSentence:(id)sender {
    /*
     * We have ended a sentence. Write some files that go with this sentence
     */
    NSLog(@"Sentence end!");
    SENTENCE_START = NO;
```

```
    // write the files with number; may be more than one sentence
    writeFile(a_left, ai_left, [NSString
     ↪  stringWithFormat:@"sentence_%d_left_acceleration.csv", SENT_NUMBER]);
    writeFile(o_left, oi_left, [NSString
     ↪  stringWithFormat:@"sentence_%d_left_orientation.csv" , SENT_NUMBER]);
    writeFile(a_right, ai_right, [NSString
     ↪  stringWithFormat:@"sentence_%d_right_acceleration.csv", SENT_NUMBER]);
    writeFile(o_right, oi_right, [NSString
     ↪  stringWithFormat:@"sentence_%d_right_orientation.csv" , SENT_NUMBER]);
    SENT_NUMBER += 1;
}

// Include some global variables; we have to split this up into many functions
// that are called at a rate of 50Hz and can't really have a variable passed
// into them (plus, where do they return to?)
//
int WORD_SAMPLE_SEP = 100; // how many instances of each word do we want
int ZERO_INDEX = 0; // (unneeded!) sees how long the pause has been happening (unneeded!)
float THRESHOLD = 0.0; // (unneeded!) our acceleration magnitude should be less than this
int DICT_INDEX = 0; // how many words have we signed?
bool WORD_SEP_HAPPENED_ONCE = NO; // to play the tone only once
bool SIGNING_WORD = NO; // are we signing a word?
int DICT_NUMBER = 0; // how many dictionary numbers?
int SENT_NUMBER = 0; // how many sentences?
bool DICT_START = NO;
bool SENTENCE_START = NO;
int N_WORDS = 100;

// left acceleration `a`, orientation `o` and indexes `oi` and `ai`
float * a_left;
float * o_left;
int oi_left;
int ai_left;

// right acceleration `a`, orientation `o` and indexes `oi` and `ai`
float * a_right;
float * o_right;
int oi_right;
int ai_right;

// how large are the vectors?
//   1min * 60 s * 50Hz
```

```
int N = 8.5 * 50 * 3;

- (void)didReceiveOrientationEvent:(NSNotification *)notification {
    /*
     * We have received an orientation event. Get the orientations and put them
     * in the vector/array
     */
    // Retrieve the orientation from the NSNotification's userInfo with the
     ↪  kTLMKeyOrientationEvent key.
    TLMOrientationEvent *orientationEvent = notification.userInfo[kTLMKeyOrientationEvent];
    TLMMyo* myo = orientationEvent.myo;

    // Create Euler angles from the quaternion of the orientation.
    TLMEulerAngles *angles = [TLMEulerAngles
     ↪  anglesWithQuaternion:orientationEvent.quaternion];

    // Next, we want to apply a rotation and perspective transformation based on the pitch,
     ↪  yaw, and roll.
//    CATransform3D rotationAndPerspectiveTransform =
 ↪  CATransform3DConcat(CATransform3DConcat(CATransform3DRotate (CATransform3DIdentity,
 ↪  angles.pitch.radians, -1.0, 0.0, 0.0), CATransform3DRotate(CATransform3DIdentity,
 ↪  angles.yaw.radians, 0.0, 1.0, 0.0)), CATransform3DRotate(CATransform3DIdentity,
 ↪  angles.roll.radians, 0.0, 0.0, -1.0));

    float o_x = angles.pitch.radians;
    float o_y = angles.yaw.radians;
    float o_z = angles.roll.radians;

    if (DICT_START && SIGNING_WORD){
        if ([myo.name isEqualToString:@"Gesture 1"]){
            oi_left = assignEventValues(o_left, oi_left, o_x, o_y, o_z);
        }
        if ([myo.name isEqualToString:@"Gesture 2"]){
            oi_right = assignEventValues(o_right, oi_right, o_x, o_y, o_z);
        }
    }
    if (SENTENCE_START){
        if ([myo.name isEqualToString:@"Gesture 1"]){
            oi_left = assignEventValues(o_left, oi_left, o_x, o_y, o_z);
        }
        if ([myo.name isEqualToString:@"Gesture 2"]){
            oi_right = assignEventValues(o_right, oi_right, o_x, o_y, o_z);
        }
```

```
    }
}

int assignEventValues(float* vector, int index, float x, float y, float z){
    /*
     * We need to assign to vectors a lot; let's wrap it in a function
     */
    vector[index+0] = x;
    vector[index+1] = y;
    vector[index+2] = z;
    index = index + 3;
    return index;
}
-(void)startWord{
    /*
     * We have started a word
     */
    SIGNING_WORD = YES;
    ZERO_INDEX = 0;
    WORD_SEP_HAPPENED_ONCE = NO;
    [self playBeep];
}

- (void)didReceiveAccelerometerEvent:(NSNotification *)notification {
    /*
     * We have received an acceleration event; put the numbers into the
     * appropriate vector/index
     */
    // Retrieve the accelerometer event from the NSNotification's userInfo with the
    ↪  kTLMKeyAccelerometerEvent.
    TLMAccelerometerEvent *accelerometerEvent =
    ↪  notification.userInfo[kTLMKeyAccelerometerEvent];
    TLMMyo* myo = accelerometerEvent.myo;
    // have two if statements for assigning left/right vectors
    // include WORD_SAMPLE_SEP*2
    // that surprisingly easy, right?

    // Get the acceleration vector from the accelerometer event.
    GLKVector3 accelerationVector = accelerometerEvent.vector;

    // Calculate the magnitude of the acceleration vector.
    float mag = GLKVector3Length(accelerationVector);
```

```
        float a_x = accelerationVector.x;
        float a_y = accelerationVector.y;
        float a_z = accelerationVector.z;
280
        if (SENTENCE_START){
            if ([myo.name isEqualToString:@"Gesture 1"]){
                ai_left = assignEventValues(a_left, ai_left, a_x, a_y, a_z);
            }
285         if ([myo.name isEqualToString:@"Gesture 2"]){
                ai_right = assignEventValues(a_right, ai_right, a_x, a_y, a_z);
            }
        }
        if (DICT_START){
290
            if(mag < THRESHOLD){
                if (ZERO_INDEX <= WORD_SAMPLE_SEP) ZERO_INDEX = ZERO_INDEX + 1;
            }
            if (SIGNING_WORD){
295             if ([myo.name isEqualToString:@"Gesture 1"]){
                    ai_left = assignEventValues(a_left, ai_left, a_x, a_y, a_z);
                }
                if ([myo.name isEqualToString:@"Gesture 2"]){
                    ai_right = assignEventValues(a_right, ai_right, a_x, a_y, a_z);
300             }
                if (ai_left > N && ai_right > N && oi_left > N && oi_right > N && DICT_INDEX <
                 ↪  [DICTIONARY count]){
                    writeFile(a_left, ai_left,   [NSString
                     ↪  stringWithFormat:@"%@_%d_left_acceleration",  [DICTIONARY
                     ↪  objectAtIndex:DICT_INDEX], DICT_NUMBER]);
                    writeFile(o_left, oi_left,   [NSString
                     ↪  stringWithFormat:@"%@_%d_left_orientation" ,  [DICTIONARY
                     ↪  objectAtIndex:DICT_INDEX], DICT_NUMBER]);
                    writeFile(a_right, ai_right, [NSString
                     ↪  stringWithFormat:@"%@_%d_right_acceleration", [DICTIONARY
                     ↪  objectAtIndex:DICT_INDEX], DICT_NUMBER]);
305                 writeFile(o_right, oi_right, [NSString
                     ↪  stringWithFormat:@"%@_%d_right_orientation" , [DICTIONARY
                     ↪  objectAtIndex:DICT_INDEX], DICT_NUMBER]);
                    NSLog(@"Writing word %@", [DICTIONARY objectAtIndex:DICT_INDEX]);

                    SIGNING_WORD = NO;
                    DICT_INDEX += 1;
310                 if (DICT_INDEX >= [DICTIONARY count]){
```

```
                    DICT_START = NO;
                }
                ai_left = 0;
                ai_right = 0;
                oi_right = 0;
                oi_left = 0;
                [self startWord];
            }
        }
    }
}

void writeFile(float * vector, int N, NSString* filename){
    /*
     * From stackoverflow I believe
     */
    NSString * str = @"";
    for (int i=0; i<N; i++){
        NSString * accel = [NSString stringWithFormat:@"%f,", vector[i]];
        str = [str stringByAppendingString:accel];
    }

    NSString *strPath = [NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
    ↪  NSUserDomainMask, YES) objectAtIndex:0];
    strPath = [strPath stringByAppendingPathComponent:filename];
    [str writeToFile:strPath
            atomically:NO
              encoding:NSStringEncodingConversionAllowLossy
                 error:nil];
}

@end
```